1 Progress Lemma

To finish the proof of Soundness we need to prove Progress. The Progress lemma captures the idea that we cannot get stuck when evaluating a well-formed expression.

**Progress Lemma:** \(\vdash e : \tau \Rightarrow e \in \text{Value} \lor \exists e'. e \rightarrow e'\)

**Proof:** We shall use induction on the typing derivation of \(e\). Remember the definition of an expression in \(\lambda^{-}\):

\[
e ::= b \mid x \mid \lambda x \in \tau . e \mid e_0 e_1
\]

So we have four cases:

- **Case** \(e = b\): We have that \(b \in \text{Value}\).
- **Case** \(e = x\): This case is not possible because we would have \(\vdash x : \tau\) and from the empty environment we cannot assign any type to \(x\).
- **Case** \(e = \lambda x \in \tau_0 . e_1\): We have that \(e \in \text{Value}\).
- **Case** \(e = e_0 e_1\): We know that there is a typing derivation for \(\vdash e_0 e_1 : \tau\) and this derivation must have the form:

\[
\begin{align*}
\vdash e_0 : \tau' \quad \vdash e_1 : \tau' \rightarrow \tau \\
\vdash e_0 e_1 : \tau
\end{align*}
\]

By the induction hypothesis, \(e_0 \in \text{Value} \lor \exists e'_0 . e_0 \rightarrow e'_0\) and \(e_1 \in \text{Value} \lor \exists e'_1 . e_1 \rightarrow e'_1\). We have four possibilities now:

- Both \(e_0\) and \(e_1\) are values. Since \(e_0\) has an arrow type, it has to be an abstraction. Say \(e_0\) is \(\lambda x \in \tau' . e_2\) and \(e_1\) is some value \(v\). Then

\[
e = (\lambda x \in \tau' . e_2)v \mapsto e_2\{v/x\}
\]

so, \(e' = e_2\{v/x\}\) as desired.
- \(e_0\) is not a value. Then \(\exists e'_0 . e_0 \mapsto e'_0\) and we have

\[
e_0 \mapsto e'_0
\]

\[
e_0 e_1 \mapsto e'_0 e_1
\]

- \(e_0\) is some value \(v\), but \(e_1\) is not a value. Then \(\exists e'_1 . e_1 \mapsto e'_1\) and we have

\[
e_1 \mapsto e'_1
\]

\[
v e_1 \mapsto v e'_1
\]

And this finishes the proof.

2 \(\lambda^{-+}\)

In comparison to \(uF\), the language \(\lambda^{-}\) has a lot of stuff missing. Let’s add some of this stuff to the language to make it more interesting. We extend \(\lambda^{-}\) to \(\lambda^{-+}\) as follows:

\[
e ::= \ldots \mid \langle e_0, e_1 \rangle \mid \text{left } e \mid \text{right } e \mid \text{case } e_0 \text{ of } e_1 | e_2 \mid \text{inl}_{\tau_1+\tau_2} e \mid \text{inr}_{\tau_1+\tau_2} e
\]
We also extend our values
\[
v ::= \lambda x \in \tau . e \mid \langle v_0, v_1 \rangle \mid \text{inl}_{\tau_1 + \tau_2} v \mid \text{inr}_{\tau_1 + \tau_2} v
\]
The set of types is defined by
\[
\tau ::= B \mid \tau_0 \to \tau_1 \mid \tau_0 \ast \tau_1 \mid \tau_0 + \tau_1
\]
where \( \tau_0 \ast \tau_1 \) and \( \tau_0 + \tau_1 \) are the product type and sum type of \( \tau_0 \) and \( \tau_1 \).

Now we define the Context operational semantics. We start extending our contexts:
\[
C ::= \ldots \mid \langle C, e \rangle \mid \langle v, C \rangle \mid \text{left} C \mid \text{right} C \mid \text{case} C \text{ of } e_1 | e_2 \mid \text{inl}_{\tau_1 + \tau_2} C \mid \text{inr}_{\tau_1 + \tau_2} C
\]
and then we define the rules. We have the usual rule
\[
e \mapsto e' \quad C[e] \mapsto C[e']
\]
where the redex reductions are
- \((\lambda x \in \tau . e)v \mapsto e\{v/x\}\)
- \(\text{left} \langle v_0, v_1 \rangle \mapsto v_0\)
- \(\text{right} \langle v_0, v_1 \rangle \mapsto v_1\)
- \(\text{case } (\text{inl}_{\tau_1 + \tau_2} v) \text{ of } e_1 | e_2 \mapsto e_1 v\)
- \(\text{case } (\text{inr}_{\tau_1 + \tau_2} v) \text{ of } e_1 | e_2 \mapsto e_2 v\)

Observe that we have constructors and destructors. The constructors construct elements of more complex types from simpler ones. For example \(\langle \cdot , \cdot \rangle\) constructs elements of type \(\tau_0 \ast \tau_1\) from two elements, one of type \(\tau_0\) and the other of type \(\tau_1\). The other constructors are the abstraction and the inclusions \(\text{inl}\) and \(\text{inr}\). The destructors are the application, case, left and right operations. A redex is an expression where a constructor and its corresponding destructor meet.

Also observe that we do not need booleans in \(\lambda^{++}\). They can encoded as follows:
- \([\text{bool}] = 1 + 1\)
- \([\#t] = \text{inl}_{1+1} \#u\)
- \([\#f] = \text{inr}_{1+1} \#u\)
- \([\text{if } e \text{ then } e_1 \text{ else } e_2] = \text{case } e_0 \text{ of } \lambda x [e_1] | \lambda x [e_2]\) where \(x\) is a fresh variable.

3 Typing Rules

Now we give the typing rules for typed lambda calculus:
\[
\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \langle e_0, e_1 \rangle : \tau_0 \ast \tau_1}
\]
\[
\frac{\Gamma \vdash e : \tau_0 \ast \tau_1}{\Gamma \vdash \text{left} e : \tau_0}
\quad
\frac{\Gamma \vdash e : \tau_0 \ast \tau_1}{\Gamma \vdash \text{right} e : \tau_1}
\]
\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2}
\quad
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2}
\]
\[
\frac{\Gamma \vdash e_2 : \tau_2 \to \tau_3 \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_0 : \tau_1 + \tau_2}{\Gamma \vdash \text{case } e_0 \text{ of } e_1 | e_2 : \tau_3}
\]
4 Denotational semantics

The denotational semantics for type domains are as follows:

\[
\begin{align*}
T[\tau_1 \to \tau_2] &= T[\tau_2][T[\tau_1]] \\
T[\tau_1 \times \tau_2] &= T[\tau_1] \times T[\tau_2] \\
T[\tau_1 + \tau_2] &= T[\tau_1] + T[\tau_2]
\end{align*}
\]

In the right hand side, \times and + mean mathematical product and disjoint union. Now we give the semantic function for this language:

\[
\rho \mid= \Gamma \Rightarrow C[\Gamma \vdash e : \tau] \rho \in T[\tau]
\]

\[
\begin{align*}
C[\Gamma \vdash \langle e_0, e_1 \rangle : \tau_0 \times \tau_1] \rho &= \langle C[\Gamma \vdash e_0 : \tau_0] \rho, C[\Gamma \vdash e_1 : \tau_1] \rho \rangle \in T[\tau_0 \times \tau_1] \\
C[\Gamma \vdash \text{left } e : \tau_0] \rho &= \pi_1(C[\Gamma \vdash e : \tau_0 \times \tau_1] \rho) \in T[\tau_0] \\
C[\Gamma \vdash \text{right } e : \tau_1] \rho &= \pi_2(C[\Gamma \vdash e : \tau_0 \times \tau_1] \rho) \in T[\tau_1] \\
C[\Gamma \vdash \text{in}l_{\tau_1 + \tau_2} e : \tau_1 + \tau_2] \rho &= \text{in}_1(C[\Gamma \vdash e : \tau_1] \rho) \in T[\tau_1 + \tau_2] \\
C[\Gamma \vdash \text{in}r_{\tau_1 + \tau_2} e : \tau_1 + \tau_2] \rho &= \text{in}_2(C[\Gamma \vdash e : \tau_2] \rho) \in T[\tau_1 + \tau_2] \\
C[\Gamma \vdash \text{case } e_0 \text{ of } e_1 | e_2] \rho &= \text{case } C[\Gamma \vdash e_0 : \tau_0 + \tau_2] \rho \text{ of } \\
& \quad \text{in}_1(x_1).C[\Gamma \vdash e_1 : \tau_1 \to \tau_3] \rho x_1 \\
& \quad | \text{in}_2(x_2).C[\Gamma \vdash e_2 : \tau_2 \to \tau_3] \rho x_2 \\
& \quad \text{end } \in T[\tau_3]
\end{align*}
\]

Just notice that the sums and products we gave above can be extended to arbitrary tuples. But this can be attained by desugaring:

\[
\begin{align*}
\tau_1 \ast \ldots \ast \tau_n &= \tau_1 \ast (\tau_2 \ast \ldots \ast \tau_n) \\
\langle e_1, \ldots, e_n \rangle &= \langle e_1, \langle e_2, \ldots, e_n \rangle \rangle
\end{align*}
\]

Sums can be desugared similarly.

5 Add Recursion

To make the language Turing-equivalent, extend the language as follows:

\[
e ::= \ldots \mid \text{rec } y : \tau \to \tau'.(\lambda x.e)
\]

\[
\frac{\Gamma, x : \tau, y : \tau \to \tau' \vdash e : \tau'}{\Gamma \vdash \text{rec } y : \tau \to \tau'.(\lambda x.e) : \tau \to \tau'}
\]
\[ C[\Gamma \vdash \text{rec } y : \tau \to \tau'.(\lambda x \ e) : \tau \to \tau']\rho = \text{fix } f \in T[\tau \to \tau'] . \]
\[ \lambda v \in T[\tau] C[\Gamma, x : \tau, \ y : \tau \to \tau' \vdash e : \tau']\rho[x \mapsto v, y \mapsto f] \]

Notice here we take a fixed point, so we need the domain \( T[\tau \to \tau'] \) to be a pointed cpo. So we need to add \( \bot \) to make this domain a pointed cpo:

\[ T[\tau \to \tau'] = T[\tau] \to T[\tau']_\bot \]
\[ \rho \models \Gamma \Rightarrow C[\Gamma \vdash e : \tau]\rho \in T[\tau]_\bot \]

By adding recursion to this language and making the domains to be pointed cpo, we can write non-terminating program in this language. We also have to do a few changes in the definition of \( C[\cdot] \) using the let construct from the meta-language to handle the \( \bot \)'s.

Example:

\[ C[\Gamma \vdash e_0 \ e_1 : \tau']\rho = \text{let } f = C[\Gamma \vdash e_0 : \tau \to \tau']\rho, \]
\[ \text{let } v = C[\Gamma \vdash e_1 : \tau] \rho, f(v) \]